Report

Question 1

Other solution:
For this question, one possible method is to iterate through the players and comparing their life values with all other players to determine the outcome of the battle. This method has time complexity of O(n^2).

My solution:
I used a stack and it's similar to the balanced symbol matching algorithm. By observation, we need only to handle the players that runs into each other, i.e., those who go down on the upper floors and those who go up from the lower floors. So, first sort the players by their floors, from upper floors to lower ones and then iterate through them. If the current player is going down, push them into the stack waiting for collision; if the current player is going up, we first check if there's elements in the stack, if no, then this player's safe and push it in to survivors; if yes, the top one will run into this player. If the going-up player lose, push the going-down player back with a reduced health and eliminate going-up player; if the going-up player win, eliminate the going-down player and battle with the next player in the stack (implement by recursion).This method has time complexity of O(nlog(n)), for the sorting.

My solution only compares players that run into each other using the characteristics of a stack and is much more time efficient.

Question 2
Other solution:
To calculate this problem, a rather direct way is the brute force method, where it iterates through the trench to find boundaries of each position. The steps are:
First find maximum width of each position: For each position i, iterate towards the left to find the left boundary where the depth becomes smaller and towards the right to find the right boundary where the depth becomes smaller.
Then calculate the area: For each position i, calculate the width by the boundaries, then multiply the width by the depth at position i to get the area of the rectangle.
Then return the maximum area.
Complexity: O(n^2).

My solution:
I used the brute force method at first and noticed its inefficiency to implement. I improved by introducing a stack. When Iterating through the trench from left to right, we push index of each position into stack. Suppose we're at i and the depth is depths[i]. When the stack is not empty (indicating there are elements with pending right boundaries) and the current depth depths[i] is smaller than the depth of the top element of the stack, it means we have found the right boundary for the top

element of the stack. In this case, we can pop elements from the stack until it's empty or the current element's depth is greater than the top element's depth. After popping elements from the stack, the top element's right boundary is the current position i. The left boundary of the current position i corresponds to the element that was just popped from the stack. Then we push the index of the current position i into the stack for further determination of its right boundary. Then calculate the area for each position using the width obtained from the stack (right boundary - left boundary - 1) and the depth at that position and return the maximum area. Complexity: O(n).

My method optimizes the process of finding left and right boundaries. It eliminates the need for nested loops and directly finds boundaries in a single pass through the trench. As a result, my approach has a linear time complexity of O(n), making it significantly faster, especially for large datasets.